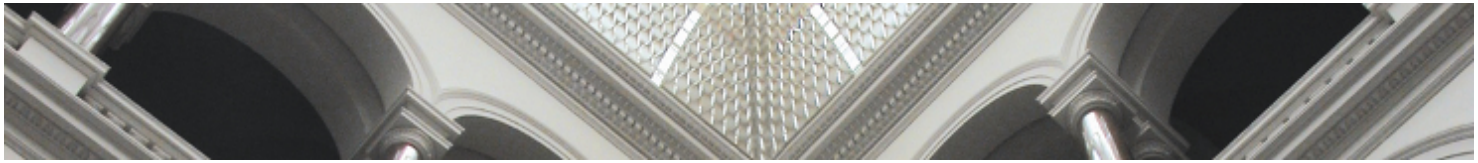
Technische Universität Berlin

# 3. Praktikumstermin

Rechnerorganisation Praktikum WiSe22/23 | Architektur eingebetteter Systeme |
Prozesse, Kontrollstrukturen, Variablen



Technische Universität Berlin

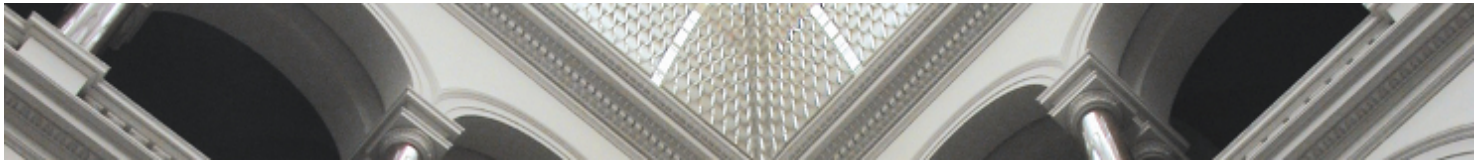
Generics

# Generischer VHDL-Code

- VHDL unterstützt generische Schnittstellen und Implementierungen
  - Implementierung kann bei der Instanziierung angepasst werden
  - Implementierung kann besser wiederverwendet werden
- Schlüsselwort `generic` in `entity` zum Deklarieren
- `generics` können in `entity` und `architecture` benutzt werden.

## Besipiel: Generisches 3-input OR-Gatter

```
entity generic3Or is
    generic ( N : integer := 16 ); -- generic N vom Typ integer mit dem Standardwert 16
    port( a : in std_logic_vector ((N - 1) downto 0); -- Verwendung des generic zur
        b : in std_logic_vector ((N - 1) downto 0);   -- Definition der Ein- und
        c : in std_logic_vector ((N - 1) downto 0);   -- Ausgangsvektoren.
        y : out std_logic_vector ((N - 1) downto 0));
end entity generic3Or;

architecture behavioral of generic3Or is
begin
    y <= a or b or c;
end architecture behavioral;
```



Technische Universität Berlin

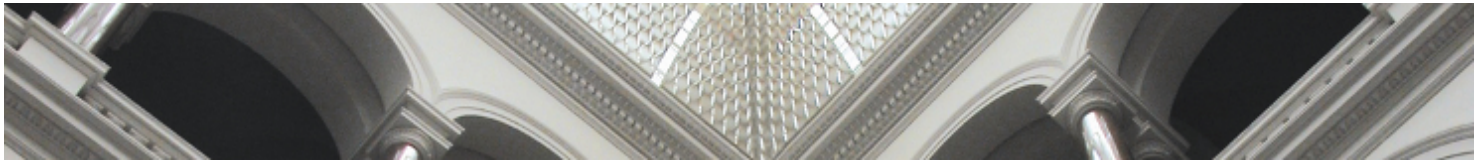
Generics

# Instanziierung

- `Generics` können bei der Instanziierung angegeben werden
- `generic map` analog zu `port map`
  - **Achtung:** kein Semikolon nach `generic map`

## Besipiel: Instantiierung des generischen 3-input OR-Gatters

```
    signal in1,in2, in3  : std_logic_vector(41 downto 0) := (others => '0');
    signal output        : std_logic_vector(41 downto 0);
...
    or42: entity work.genericOr3
        generic map(     -- Zuweisung der Generics
            N => 42      -- Instanziierung mit N = 42
        )                -- Kein Semikolon
        port map(        -- Zuweisung von In- und Outputs
            a => in1,
            b => in2,
            c => in3,
            y => output
        );
...
```



Technische Universität Berlin

Verhaltensebene

# Nebenläufige Anweisungen

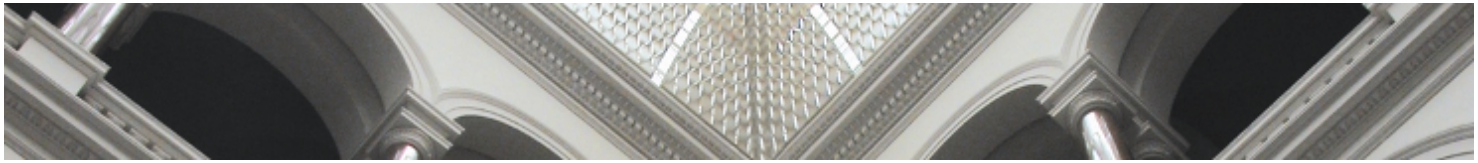
- VHDL-Code wird standardmäßig *nebenläufig* interpretiert
- d. h. der Code wird nicht (wie z. B. in C) nacheinander, von oben nach unten, sondern vollständig zeitgleich abgearbeitet
- Die Reihenfolge hat *keinen* Einfluss auf das Ergebnis:

```
architecture foo1 of bar is
begin
  a <= '1'
  b <= '0'
  y <= a and b
end;
```

```
architecture foo2 of bar is
begin
  y <= a and b
  b <= '0'
  a <= '1'
end;
```

*Beide Beschreibungen führen zu demselben Ergebnis*

- **VHDL-Schlüsselwörter:**
  *when-else*, *with-select*, . . .



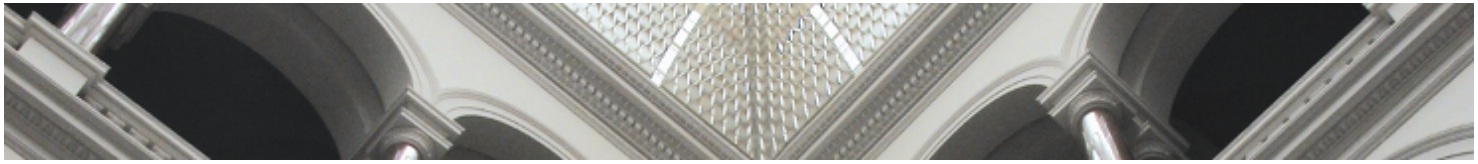
Technische Universität Berlin

Verhaltensebene

# Sequenzielle Anweisungen

- Deshalb existieren in VHDL sog. Prozess-Statements
- Anweisungen in einem Prozess werden sequenziell ausgeführt
- Das Prozess-Statement selbst ist aber nebenläufig
- Es können also mehrere Prozesse parallel ablaufen

- **VHDL-Schlüsselwörter:**
  *process*, *if-else-elsif*, *case-when*, *function*, *procedure*, . . .



Technische Universität Berlin

# Prozess-Statement

– Ein Prozess kann *nur* im Anweisungsteil einer `architecture` stehen

```
[<process_name>: ]process (<sensitivity_list>)
  -- space for variable declarations
begin
  -- sequential statements
end process;
```

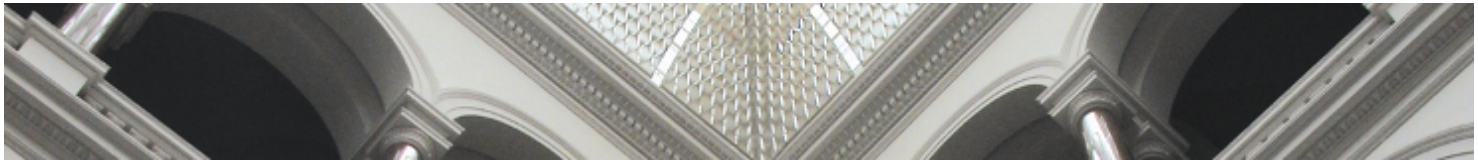
– Der Prozess-Name ist optional und kann weggelassen werden

**Empfindlichkeitsliste (engl. sensitivity list)**

– Liste der Signale, bei denen der Prozess aktiviert wird

– Änderung eines Signals bewirkt Abarbeitung des Prozesses

– Faustregel für **kombinatorische** Prozesse: alle Signale, auf die innerhalb des Prozesses lesend zugegriffen wird (Signalzuweisung, `case`, `if`), sollten in der Empfindlichkeitsliste stehen!



Technische Universität Berlin



# Was ist das?

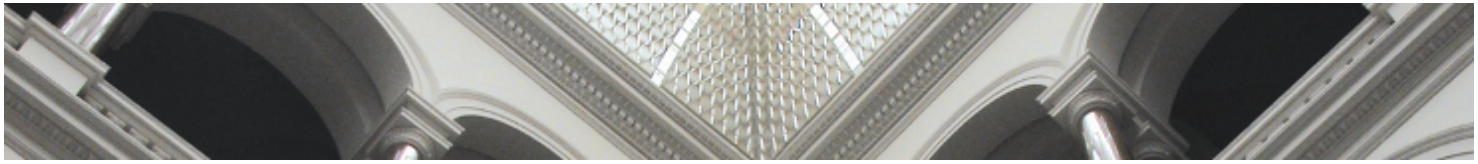
## Kontrollstrukturen...

- ... fassen mehrere Anweisungen in einer zusammen
- ... können demnach situationsabhängig „entscheiden“

- Es existieren zwei verschiedene Kontrollstrukturen
- Wir demonstrieren diese im Folgenden anhand der Entity:

```
entity bcd is
    port(bcd     : in  std_logic_vector(3 downto 0);
         bitmask : out std_logic_vector(6 downto 0));
end entity bcd;
```

- Außerdem zeigen wir ihr jeweiliges nebenläufiges Pendant



Technische Universität Berlin

Kontrollstrukturen

# Gegenüberstellung I

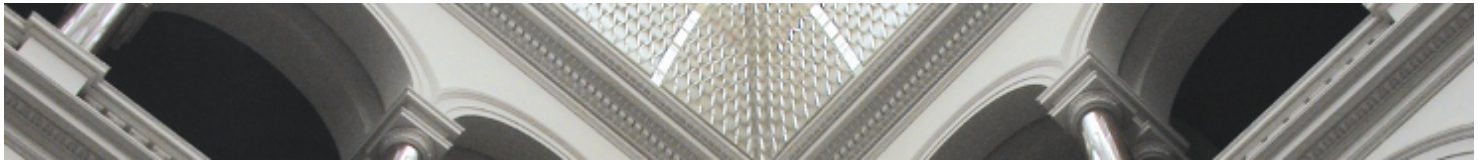
## Sequenziell:

```
architecture if_example of bcd is
begin
  bcd_to_7seg: process(bcd)
  begin
    if (bcd = "0000") then
      bitmask <= "0111111";
    elsif (bcd = "0001") then
      bitmask <= "0000110";
    -- ...
    else
      bitmask <= "1000000";
    end if;
  end process;
end architecture if_example;
```

## Nebenläufig:

```
architecture when_example of bcd is
begin
  output <= "0111111" when bcd = "0000" else
            "0000110" when bcd = "0001" else
            -- ...
            "1000000";
end;
```

- *Die Beispiele wurden aus Platzgründen gekürzt*
- **für priorisierte Abfragen**:
- Erste erfüllte Bedingung beendet den Prozess
- Nützlich, wenn mehrere Belegungen möglich sind
- Komplexere Bedingungen, z.B. `A and B` ebenfalls möglich



Technische Universität Berlin

Kontrollstrukturen

# Gegenüberstellung II

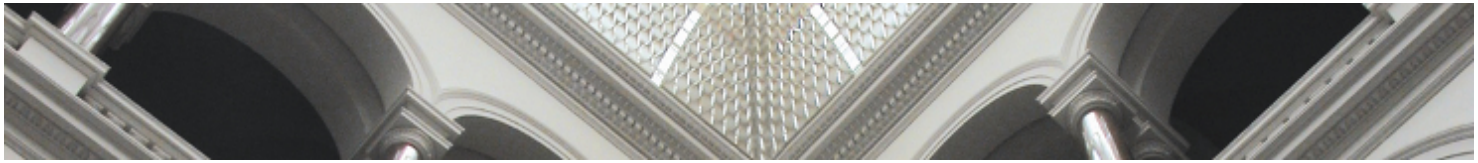
## Sequenziell:

```
architecture case_example of bcd is
begin
  bcd_to_7seg: process(bcd)
  begin
    case bcd is
      when "0000" => bitmask <= "0111111";
      when "0001" => bitmask <= "0000110";
      -- ...
      when others => bitmask <= "1000000";
    end case;
  end process;
end architecture case_example;
```

## Nebenläufig:

```
architecture with_example of bcd is
begin
  with bcd select
    bitmask <= "0111111" when "0000",
               "0000110" when "0001",
              -- ...
               "1000000" when others;
end architecture with_example;
```

- *Die Beispiele wurden aus Platzgründen gekürzt*
- **Für exklusive Abfragen**:
- Nützlich, wenn bloß eine Belegung möglich ist
- Alle anderen Bedingungen notwendigerweise falsch
- Komplexere Bedingungen, z.B. `A and B` ebenfalls möglich



Technische Universität Berlin

# Variablen

- Innerhalb eines Prozesses können *Variablen* deklariert werden

```
variable <variable_name>: <data_type>;                    -- Syntax

variable test_vector: std_logic_vector(3 downto 0);       -- Beispiel
```

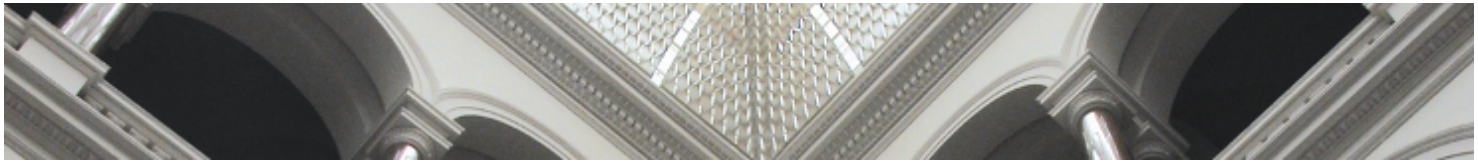
- Ihnen kann genauso wie Signalen ein Wert zugewiesen werden

```
<variable_name> := <value>;                               -- Syntax

test_vector := "1000";                                    -- Beispiel
```

- Die Zuweisung erfolgt bei Variablen mit `:=`, bei Signalen mit `<=`
- Wir können auch bei der Deklaration mit einem Wert initialisieren:

```
variable <variable_name>: <data_type> := <value>;         -- Syntax 1
signal <signal_name>: <data_type> := <value>;             -- Syntax 2

variable test_vector1: std_logic_vector(3 downto 0) := "1000";   -- Beispiel 1
signal   test_vector2: std_logic_vector(3 downto 0) := "0110";   -- Beispiel 2
```

- Die Initialisierung erfolgt *immer* mit `:=`



Technische Universität Berlin

# Variablen

- Bei Variablen erfolgt diese Zuweisung aber *sofort*
- Bei Signalen hingegen erst zum *Ende* des Prozesses

```
entity bar is
  port(i:      in  std_logic;
       o1, o2: out std_logic);
end entity bar;

architecture foo1 of bar is
begin
  process (i)

  begin
    o1 <= i;

    o2 <= not o1;
    -- o2 != not i
  end process;
end architecture foo1;
```

```
entity bar is
  port(i:      in  std_logic;
       o1, o2: out std_logic);
end entity bar;

architecture foo2 of bar is
begin
  process (i)
    variable tmp: std_logic;
  begin
    o1 <= i;
    tmp := i;
    o2 <= not tmp;
    -- o2 == not i
  end process;
end architecture foo2;
```

☞ Werden einem Signal innerhalb eines Prozesses mehrere Werte zugewiesen, ist *nur* die letzte Zuweisung tatsächlich gültig!